Demystifying the Power of Large Language Models in Graph Structure Generation

Yu Wang[1] **Ryan Rossi**[2] **Namyong Park** **Nesreen Ahmed**[3]
Danai Koutra[4] **Franck Dernoncourt**[2] **Tyler Derr**[5]
[1] University of Oregon [2] Adobe Research [3] Cisco AI Research
[4] University of Michigan [5] Vanderbilt University

Abstract

Despite the unprecedented success of applying Large Language Models (LLMs) to graph discriminative tasks such as node classification and link prediction, its potential for graph structure generation remains largely unexplored. To fill this crucial gap, this paper presents a systematic investigation into the capability of LLMs for graph structure generation. Specifically, we design prompts triggering LLMs to generate codes that optimize network properties by injecting domain expertise from network science. Since graphs in different domains exhibit unique structural properties captured by various metrics (e.g., clustering coefficient capturing triangles in social networks while squares reflecting road segments in transportation networks), we first evaluate the capability of LLMs to generate graphs satisfying each structural property in different domains. After that, we select the optimal property configurations and benchmark the graph structure generation performance of LLMs against established graph generative models across multiple domains. Our findings shed light on generating graph structures from an LLM perspective. Our code is publically available here.

1 Introduction

Large Language Models (LLMs) have achieved remarkable success in various real-world applications (Wu et al., 2024; Wang et al., 2024; Han et al., 2024), including graph-based tasks such as node classification and link prediction (Yan et al., 2023; Chen et al., 2024). Two prominent paradigms for enhancing graph tasks through LLMs are LLM-as-Enhancers/Predictors. For the first paradigm (He et al., 2023), LLMs are used to enrich graph data by leveraging their pre-trained knowledge to enhance the node/edge textual attributes. In the second paradigm (Chen et al., 2023b), LLMs are used to encode textual information for each node/edge and directly participate in downstream predictions. Although both approaches have shown significant success, they primarily focus on discriminative tasks, leaving a wide gap in graph-generative tasks.

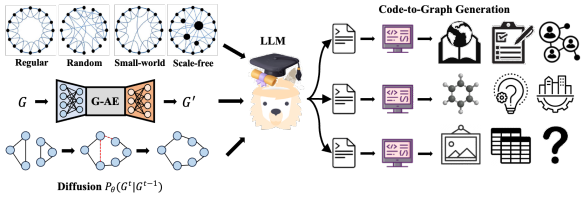


Figure 1: As pre-trained LLMs store knowledge about graph structures and graph generative models, we prompt LLMs with prescribed graph properties to generate codes that can further generate graphs.

Unlike discriminative tasks, graph generation aims to produce graphs of target distributions (Livernoche et al., 2023; Liu et al., 2024). Given the wide variation in graph structural properties of different domains, such as the prevalence of triangles in social networks versus squares in road networks (Rossi and Ahmed, 2019), graph generators must understand different structural patterns and adapt accordingly to the target domains. To capture these diverse structural patterns, numerous graph generative models have been developed, including heuristic-based ones (Goldenberg et al., 2010; Kolaczyk and Csárdi, 2014) and deep learning-based ones (Guo and Zhao, 2022; Zhu et al., 2022). Heuristic-based methods utilize pre-defined rules to generate graphs approaching the target observations. However, its hard-coded rule oversimplifies the complex distribution of real-world graphs. Deep learning-based graph generative models automatically capture intricate statistics by learning to recover graphs from training data (Simonovsky and Komodakis, 2018; You et al., 2018; Zang and Wang, 2020). Despite their effectiveness, they are heavily based on training data and hardly generalize to unseen domains (**?**). Furthermore, the most state-of-the-art diffusion-based graph generative models, such as DiGress (Vignac et al., 2022) and its scalable version SaGess (Limnios et al., 2023), can only scale up to graphs with a few thousand nodes, which are still much smaller than real-world networks (Chen et al., 2023a) such as social networks and citation networks with millions of nodes (Hu et al., 2020).

Findings of the Association for Computational Linguistics:
NAACL 2025, pages 8189–8204

Given the success of LLMs in graph discriminative tasks and the limitations of existing graph-generative models, we aim to explore the potential of LLMs in understanding and generating graph structures. Specifically, we prompt LLMs with prescribed graph structure properties to produce code that generates graphs for different domains. Our contributions are summarized as follows:

- **Demystifying the power of LLMs to understand and generate graph structures**: We investigate how well LLMs can understand graph structural properties from different domains.
- **Proposing an LLM-based Graph Generative Model**: We propose two systematic prompts to trigger LLMs to generate graphs with prescribed structure properties, one solely by LLMs and the other fusing the expertise from network science.
- **Benchmarking with real-world graphs**: We benchmark our proposed method with different baselines in generating real-world graphs from three different domains and derive new insights on LLM-based graph generative models.

2 Related Work

2.1 Large Language Model for Graph Tasks

Recently, a wide variety of graph-based tasks have been enhanced by LLMs. For discriminative tasks (Chen et al., 2024; Pan et al., 2024), the pre-trained knowledge in LLMs is either used to enhance textual attributes or directly predict node/edge classes (He et al., 2023; Chen et al., 2023b). For generative tasks, one recent study has explored the potential of LLMs for graph generation. However, it only considers very basic topologies such as trees and cycles (Yao et al., 2024). In contrast, our work focuses on real-world networks. Our work is also aligned with recent studies simulating social networks through LLM agents (Chang et al., 2024; Gao et al., 2023). However, we also consider molecule and citation networks.

2.2 Graph Generative Models

Graph generative models have been developed for graphs across various applications, such as molecular design for high drug-likeness and imperceptible adversarial attacks (Hoogeboom et al., 2022; Livernoche et al., 2023; Kang et al., 2024; Liu et al., 2024). These models can be grouped into two categories: statistical ones (Goldenberg et al., 2010; Kolaczyk and Csárdi, 2014), which rely on prescribed rules and sampling techniques to simulate networks with specific properties, and deep learning ones (Guo and Zhao, 2022; Zhu et al., 2022; **?**), which learn complex structural patterns from training graphs. Although statistical models are easier to implement, they are too hard-coded to generalize to complex real-world graphs. Deep learning ones, in contrast, are more adaptable but require more training data, and even the most advanced diffusion models, such as DiGress (Vignac et al., 2022) and its scalable version SaGress (Limnios et al., 2023), are limited to handling networks with only a few thousand nodes. Given previous deficiencies, our work explores the new paradigm of using LLMs to generate graphs.

Prompt 1: LLM-based Graph Generation

System Message: You are a network generator who is using the Python package NetworkX to write Python code to generate a network with the user-specified property. Please write a code to generate an undirected network with the following properties.

Properties:
{*Property 1 - Value 1*}, {*Property 2 - Value 2*}, ..., {*Property K - Value K*}.

Methods:
{*Property 1 - Method 1*}, {*Property 2 - Method 2*}, ..., {*Property K - Method K*}.

Instructions:
{*Instruction 1*}, {*Instruction 2*}, ..., {*Instruction K*}.

3 Method

To instruct LLMs to generate graphs with specific structural properties, we design **Prompt 1**, which includes a system message outlining the LLM's general role and three key inputs: *Properties*, *Methods*, and *Instructions*. The *Properties* specifies graph structural properties with prescribed properties extracted from target graphs that the generated graphs should approximate. The *Methods* details algorithms the LLM can use to achieve these properties. The *Instructions* provides general guidelines for the LLM to follow. To assess the LLMs' ability to understand graph structure, we propose a simplified variant of Prompt 1 by removing the *Methods*, forcing LLMs to rely solely on their own knowledge to generate graphs. We refer to this basic version as Text2Graph generation (**T2G**), while the more advanced version with *Methods* as the hint being **T2G**$^+$. Two examples of T2G/T2G$^+$ prompts and their generated codes are in Appendix A.2.

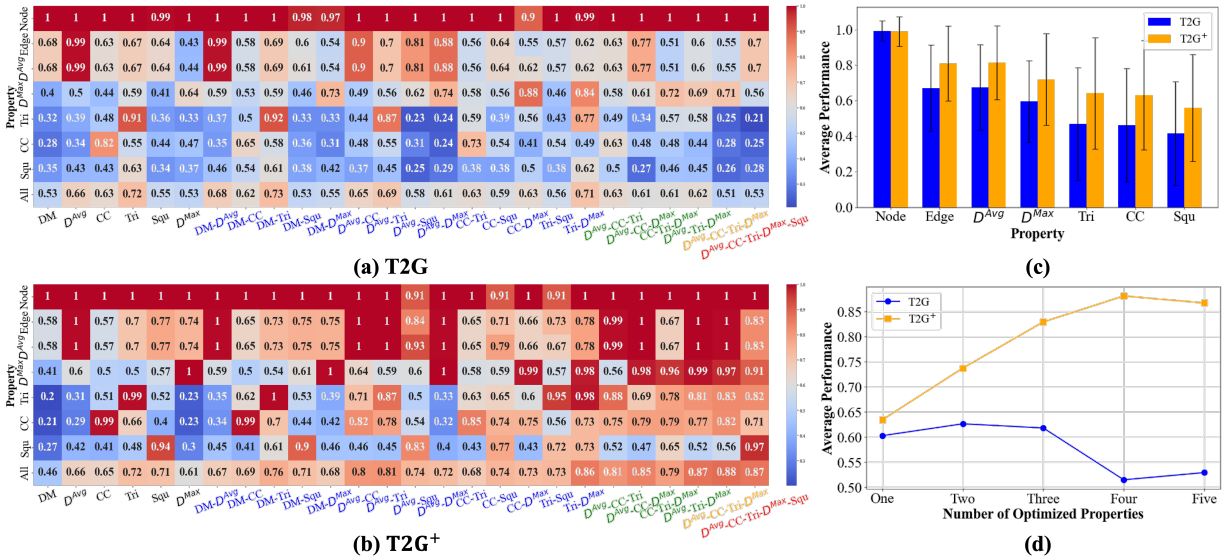


Figure 2: **(a)/(b)**: Average performance of generating graphs across all domains using T2G/T2G$^+$ by prescribing different structural properties; **(c)**: T2G/T2G$^+$ perform better in recovering lower-order graph structural properties (e.g., Node) while worse in recovering higher-order graph structural properties (e.g., Square (Squ)); **(d)**: Generative performance first increases and then decreases as the number of prescribed graph properties increases, with the changing point coming earlier on T2G compared with T2G$^+$. Additional results are in Appendix A.4

For the input of *Properties*, we explore the potential of LLMs in understanding the following six properties that comprehensively characterize graph structures in different aspects: Domain Name (DM), average degree (D^{Avg}), clustering coefficient (CC), maximum degree (D^{Max}), and the total number of triangles (Tri) and squares (Squ). The definitions and calculations for these properties are provided in Appendix A.1. We omit the number of edges, as it is naturally determined once the number of nodes and the average degree are specified.

4 Experiment

In this section, we empirically evaluate the performance of LLMs in understanding six predefined structural properties and generating graphs from three domains. For evaluation, we compute the percentage difference in six structural properties between generated graphs and ground-truth graphs. The final performance is $(1 -$ percentage difference$)$, ensuring that higher scores indicate better generation performance. Detailed experimental setting is in Appendix A.3.

Finding 1 - Figure 2(a)/(b): The performance in block (i, j) represents the ability to reconstruct the ith property when the jth property is prescribed in the prompt. According to the first six columns in Figure 2(a), LLMs possess and can leverage their internal knowledge to reconstruct graphs of each prescribed property, such as CC (82%), Tri (91%) and D^{Avg} (99%). However, LLMs are difficult to understand and generate higher-order structures such as Squ (34%). This situation is significantly improved after introducing network science expertise by specifying concrete methods for LLMs' usage in graph generation, as shown in Figure 2(b) where the performance of reconstructing square metric Squ increases significantly to 94%. This indicates that fusing domain expertise from network science enhances LLMs' understanding and generative performance on graphs, especially for higher-order metrics. Furthermore, providing domain information does not improve graph generation, evidenced by lower graph generation performance in the 1st column, suggesting that LLMs do not internally possess stereotypes about network domain knowledge and their structural properties. However, we observe that for molecular datasets like MUTAG (Listing 3 in Appendix A.2) - especially when the average degree is low - the LLM tends to initialize graphs with cycles. In contrast, most social networks are initialized by the Barabasi-Albert model (Barabási and Albert, 1999).

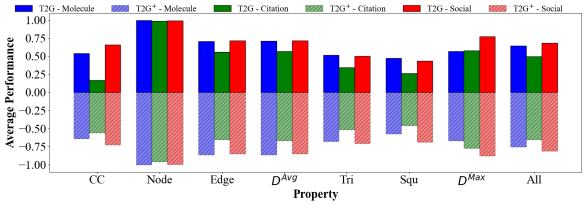


Figure 3: Generative performance in different domains, Citation, Social, and Molecule, by T2G and T2G$^+$.

Table 1: Comparison of graph generation performance across different methods. The best performance is **bolded**, and the second best is underlined. We omit the "Square (Squ)" metric on large-size REDD due to excessively long runtime. CITE - Citation Graphs, MOL - Molecule Graphs, SOC - Social Graphs.

DM	Data	Avg-Deg (D^{Avg})							Triangle (Tri)							Clustering Coefficient (CC)						
		BA	SW	ER	SF	CF	T2G	T2G$^+$	BA	SW	ER	SF	CF	T2G	T2G$^+$	BA	SW	ER	SF	CF	T2G	T2G$^+$
CITE	Cora	65.0	51.3	96.4	94.9	99.8	68.4	**100**	20.0	0.0	0.7	66.2	5.6	**99.1**	92.7	5.7	0.0	0.6	52.2	2.5	68.0	**99.9**
	Cite	68.4	73.1	98.0	77.0	99.8	99.9	**99.9**	6.3	0.0	0.1	40.3	3.3	49.4	**67.8**	5.5	0.0	0.3	95.0	1.9	**99.2**	88.6
	Pub	56.2	89.0	**99.8**	84.0	**99.8**	99.1	75.7	11.1	**86.6**	0.1	64.4	4.9	1.4	64.8	5.9	16.1	0.3	**50.0**	2.7	4.5	45.9
MOL	MUT	85.2	90.6	70.5	93.1	97.5	87.3	**100**	**100**	91.1	53.3	0.0	62.2	97.4	**100**	**100**	91.1	53.3	0.0	62.2	97.4	**100**
	PROT	90.0	67.5	82.3	84.7	94.9	84.1	**99.3**	57.2	21.3	50.9	61.2	13.0	**93.2**	84.5	51.4	26.0	30.0	39.9	14.4	52.9	**75.5**
	NCI1	90.4	94.1	60.5	78.1	97.7	91.7	**100**	**100**	97.8	66.7	0.0	71.1	**100**	**100**	**100**	97.8	66.7	0.0	71.1	**100**	**100**
	DD	84.8	79.9	83.6	91.4	99.4	**100**	99.7	30.4	48.5	11.7	**88.0**	2.5	51.3	53.7	17.7	**87.5**	6.2	45.4	2.7	77.6	51.3
	ENZY	66.5	76.2	80.4	68.9	95.5	78.6	**99.1**	58.1	46.7	49.7	34.3	19.6	**86.5**	77.9	59.6	52.0	31.1	32.2	20.4	53.5	**67.0**
SOC	IMDB	68.6	89.5	87.1	32.3	77.0	**99.7**	97.5	56.1	54.8	76.5	4.2	22.3	65.4	**85.8**	54.4	64.7	66.5	20.9	41.7	61.9	**75.0**
	REDD	60.2	86.5	52.0	70.2	89.2	34.5	**99.8**	48.8	2.0	22.3	7.9	**69.2**	10.1	56.5	57.4	4.7	9.5	23.7	**76.6**	56.5	48.9

DM	Data	Square (Squ)							Max-Deg (D^{Max})							Average						
		BA	SW	ER	SF	CF	T2G	T2G$^+$	BA	SW	ER	SF	CF	T2G	T2G$^+$	BA	SW	ER	SF	CF	T2G	T2G$^+$
CITE	Cora	**66.8**	0.0	0.7	4.7	11.9	53.7	39.1	95.8	3.0	7.1	22.7	96.4	94.1	**100**	50.7	10.9	21.1	48.1	43.2	76.6	**86.3**
	Cite	**8.8**	0.0	0.1	5.2	2.2	3.6	7.4	81.1	5.0	9.1	6.9	**98.0**	**100**	**100**	34.0	15.6	21.5	44.9	41.0	70.4	**72.7**
	Pub	11.7	**16.1**	0.3	7.8	4.6	0.7	4.4	37.6	4.7	9.4	2.7	98.8	53.9	**100**	24.5	42.5	22.0	41.8	42.2	31.9	**58.2**
MOL	MUT	**100**	95.6	57.8	51.1	53.3	94.7	42.5	46.1	84.8	73.9	44.5	**100**	87.1	**100**	86.3	90.6	61.8	37.8	75.0	**92.8**	88.5
	PROT	63.2	13.0	56.8	43.6	22.7	49.2	**68.6**	43.9	75.1	71.1	37.6	**94.4**	53.5	91.9	61.2	40.6	58.2	53.4	47.9	66.6	**83.2**
	NCI1	**100**	95.6	64.4	2.2	60.0	83.9	76.7	42.5	85.6	68.5	23.4	99.4	83.9	**100**	86.6	94.2	65.4	20.7	79.9	91.9	**100**
	DD	**68.7**	23.5	28.7	15.8	6.5	52.7	44.0	19.8	58.9	73.5	10.4	**100**	45.0	95.3	44.3	59.7	40.7	50.2	42.2	58.9	**68.0**
	ENZY	26.3	31.3	60.4	48.8	26.8	50.4	**77.9**	39.2	77.7	65.9	33.2	**95.2**	58.7	90.6	49.9	56.8	57.5	43.5	51.5	65.5	**82.9**
SOC	IMDB	50.1	48.9	73.0	3.0	20.8	73.2	**89.1**	92.1	64.6	79.3	68.0	70.7	79.2	**99.2**	64.3	64.5	76.5	25.7	46.5	75.9	**89.3**
	REDD	—	—	—	—	—	—	—	26.1	3.3	9.5	83.2	**98.3**	10.1	69.2	48.1	24.1	32.9	46.3	**75.1**	34.2	75.9

Finding 2 - Figure 2(c): For each metric, we average the graph generation performance across all property configurations. Lower-order structural properties end up with higher performance compared to higher-order properties because higher-order structural properties require more network science expertise to implement and are inherently more challenging for LLMs to satisfy. For instance, ensuring zero-order properties, such as the number of nodes, can be achieved by directly specifying the input, while higher-order properties, such as CC and Squ, demand more network science knowledge to adjust graph structures. For example, manipulating CC requires adding edges among the same node's neighbors, whereas manipulating Squ involves adding edges to close paths of length 3.

Finding 3 - Figure 2(d): As more structural properties are prescribed, the performance of graph generation initially improves but then decreases, with T2G showing this drop off earlier than T2G$^+$. We hypothesize that the initial improvement stems from the richer structure information provided to LLMs. However, imposing too many constraints narrows the space of feasible graphs, reducing the likelihood of matching the target ground truth. Moreover, graph structural properties are often interdependent—adjusting one property can inadvertently alter others. For instance, enforcing a lower average degree constraint requires removing edges, which conflicts with matching the high CC goal, where additional edges are needed. The earlier decline in T2G compared to T2G$^+$ suggests that integrating network science expertise improves LLMs' capability in balancing the competing demands of satisfying different structural properties.

Finding 4 - Figure 3: After averaging results within the same domain, we find that LLMs consistently exhibit lower performance on citation networks across all metrics. We hypothesize that this disparity arises because Cora and Citeseer are generally larger than datasets in other domains. Consequently, adding or removing edges may be more challenging to manipulate their graph structures to the desired level. Future research could investigate the disparity of LLMs in graph generation across different domains.

Finding 5 - Table 1: Based on our previous demonstration of LLMs' ability to capture certain structural properties, we now evaluate their effectiveness in generating realistic graphs. With the optimal structural configuration identified in Figure 2(a)/(b), we generate graphs and benchmark them against those generated from conventional heuristic-based models (more details are in Appendix A.3). Recent deep-learning-based models are excluded due to their inability to scale to networks larger than a few thousand nodes. As shown in Table 1, T2G$^+$ significantly outperforms

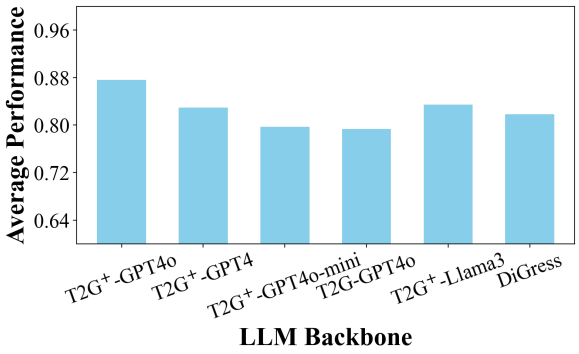


Figure 4: Graph Generation Performance across different LLM backbones and the representative deep learning-based model, DiGress.

all other methods by integrating network science expertise. Although T2G ranks lower, it still remains among the top approaches, underscoring the capacity of LLMs to capture network structures to a certain extent. In addition to the global evaluation, we provide a fine-grained visualization of DEG and CC distributions in Appendix A.4.

Finding 6 - Figure 4: To evaluate the generalizability of our T2G/T2G$^+$ framework, we include multiple LLMs, including GPT-4o, GPT-4, GPT-4o-mini, and LLaMA-3, with their performance computed by averaging results across all evaluation metrics and datasets. Compared to heuristic-based methods Table 1, both T2G and T2G$^+$ consistently outperform these baselines, demonstrating the broad applicability of our approach across different LLM backbones. Among the tested LLMs, T2G$^+$-GPT-4o achieves the best performance, and GPT-4o-mini ranks lowest, aligning with the general expectation. Additionally, we compare against DiGress, a state-of-the-art deep-learning-based graph generation model. While DiGress shows competitive performance, T2G$^+$—which incorporates domain expertise—achieves superior results.

5 Conclusion

This work explores the capability of LLMs in graph generation tasks by specifically investigating their understanding of graph structural properties and reconstructing real-world graphs. Our findings reveal that LLMs possess a certain level of understanding of graph properties, and this understanding can be significantly boosted by incorporating domain expertise of network science. Furthermore, the complexity of understanding and generating structural properties varies as the order of topological metrics, the number of constraints, and the domain of the target graph changes.

6 Limitations and Future Work

Analysis Limitations: Our proposed baseline, T2G, does not demonstrate a significantly higher graph generation performance. This indicates that LLMs still struggle to effectively leverage their internal knowledge to understand graph structural properties and generate corresponding codes.

Method Design Limitations: Our current LLM-based graph generation only supports generating graph structures. However, real-world graphs usually possess attributes such as social profiles in social networks and atom numbers for molecule graphs. This constraint could be removed in the future to make the method more generalizable.

Potential Risks: One potential risk of our LLM-based graph generation lies in its generation efficiency being maliciously used by third parties. If adversarially compromised, LLMs could be exploited to generate anomaly social graphs, posing substantial risks. However, given the current performance of this approach is still in its early stages, the immediate threat remains limited. Nonetheless, this concern highlights the need for research on adversarial graph generation, fostering a deeper understanding of potential risks and the development of mitigation strategies for future advancements.

LLM-based graph generation presents a promising direction due to its scalability and pre-trained knowledge about network formation. Traditional deep learning-based graph generative models struggle with large-scale networks, as even the most scalable graph diffusion models are limited to handling networks with up to 10,000 nodes (Chen et al., 2023a; Limnios et al., 2023), whereas real social networks often comprise millions of nodes. In contrast, LLM-based methods effectively overcome these scalability issues. Recent studies have demonstrated that LLM-powered agents can simulate social networks with structural properties resembling real graphs compared to previous baselines, highlighting LLMs' inherent understanding of graph structures. Looking ahead, we envision graph generation evolving into two paradigms: microscopic graph generation, which will continuously rely on deep learning models like graph diffusion to capture fine-grained structures in applications such as drug discovery and bioinformatics, and macroscopic graph generation, where LLM-driven approaches, particularly agent-based simulations, will dominate large-scale social network modeling (Gao et al., 2023; Chang et al., 2024).

References

Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science*, 286(5439):509–512.

Béla Bollobás, Christian Borgs, Jennifer T Chayes, and Oliver Riordan. 2003. Directed scale-free graphs. In *SODA*, volume 3, pages 132–139. Baltimore, MD, United States.

Serina Chang, Alicja Chaszczewicz, Emma Wang, Maya Josifovska, Emma Pierson, and Jure Leskovec. 2024. Llms generate structurally realistic social networks but overestimate political homophily. *arXiv preprint arXiv:2408.16629*.

Xiaohui Chen, Jiaxing He, Xu Han, and Li-Ping Liu. 2023a. Efficient and degree-guided graph generation via discrete diffusion modeling. *arXiv preprint arXiv:2305.04111*.

Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, et al. 2024. Exploring the potential of large language models (llms) in learning on graphs. *ACM SIGKDD Explorations Newsletter*, 25(2):42–61.

Zhikai Chen, Haitao Mao, Hongzhi Wen, Haoyu Han, Wei Jin, Haiyang Zhang, Hui Liu, and Jiliang Tang. 2023b. Label-free node classification on graphs with large language models (llms). *arXiv preprint arXiv:2310.04668*.

Paul Erdos, Alfréd Rényi, et al. 1960. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci*, 5(1):17–60.

Chen Gao, Xiaochong Lan, Zhihong Lu, Jinzhu Mao, Jinghua Piao, Huandong Wang, Depeng Jin, and Yong Li. 2023. S3: Social-network simulation system with large language model-empowered agents. *arXiv preprint arXiv:2307.14984*.

Anna Goldenberg, Alice X Zheng, Stephen E Fienberg, Edoardo M Airoldi, et al. 2010. A survey of statistical network models. *Foundations and Trends® in Machine Learning*, 2(2):129–233.

Xiaojie Guo and Liang Zhao. 2022. A systematic survey on deep generative models for graph generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5370–5390.

Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. 2024. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*.

Xiaoxin He, Xavier Bresson, Thomas Laurent, Adam Perold, Yann LeCun, and Bryan Hooi. 2023. Harnessing explanations: Llm-to-lm interpreter for enhanced text-attributed graph representation learning. *arXiv preprint arXiv:2305.19523*.

Emiel Hoogeboom, Vıctor Garcia Satorras, Clément Vignac, and Max Welling. 2022. Equivariant diffusion for molecule generation in 3d. In *International conference on machine learning*, pages 8867–8887. PMLR.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133.

Mintong Kang, Dawn Song, and Bo Li. 2024. Diffattack: Evasion attacks against diffusion-based adversarial purification. *Advances in Neural Information Processing Systems*, 36.

Eric D Kolaczyk and Gábor Csárdi. 2014. *Statistical analysis of network data with R*, volume 65. Springer.

Stratis Limnios, Praveen Selvaraj, Mihai Cucuringu, Carsten Maple, Gesine Reinert, and Andrew Elliott. 2023. Sagess: Sampling graph denoising diffusion model for scalable graph generation. *arXiv preprint arXiv:2306.16827*.

Gang Liu, Eric Inae, Tong Zhao, Jiaxin Xu, Tengfei Luo, and Meng Jiang. 2024. Data-centric learning from unlabeled graphs with diffusion model. *Advances in neural information processing systems*, 36.

Victor Livernoche, Vineet Jain, Yashar Hezaveh, and Siamak Ravanbakhsh. 2023. On diffusion modeling for anomaly detection. *arXiv preprint arXiv:2305.18593*.

Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review*, 45(2):167–256.

Bo Pan, Zheng Zhang, Yifei Zhang, Yuntong Hu, and Liang Zhao. 2024. Distilling large language models for text-attributed graph learning. *arXiv preprint arXiv:2402.12022*.

Ryan A Rossi and Nesreen K Ahmed. 2019. Complex networks are structurally distinguishable by domain. *Social Network Analysis and Mining*, 9:1–13.

Martin Simonovsky and Nikos Komodakis. 2018. Graphvae: Towards generation of small graphs using variational autoencoders. In *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27*, pages 412–422. Springer.

Clement Vignac, Igor Krawczuk, Antoine Siraudin, Bohan Wang, Volkan Cevher, and Pascal Frossard. 2022. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*.

Yu Wang, Nedim Lipka, Ryan A Rossi, Alexa Siu, Ruiyi Zhang, and Tyler Derr. 2024. Knowledge graph prompting for multi-document question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19206–19214.

Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442.

Zhenyu Wu, Meng Jiang, and Chao Shen. 2024. Get an a in math: Progressive rectification prompting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19288–19296.

Hao Yan, Chaozhuo Li, Ruosong Long, Chao Yan, Jianan Zhao, Wenwen Zhuang, Jun Yin, Peiyan Zhang, Weihao Han, Hao Sun, et al. 2023. A comprehensive study on text-attributed graphs: Benchmarking and rethinking. *Advances in Neural Information Processing Systems*, 36:17238–17264.

Yang Yao, Xin Wang, Zeyang Zhang, Yijian Qin, Ziwei Zhang, Xu Chu, Yuekui Yang, Wenwu Zhu, and Hong Mei. 2024. Exploring the potential of large language models in graph generation. *arXiv preprint arXiv:2403.14358*.

Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. 2018. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR.

Chengxi Zang and Fei Wang. 2020. Moflow: an invertible flow model for generating molecular graphs. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 617–626.

Yanqiao Zhu, Yuanqi Du, Yinkai Wang, Yichen Xu, Jieyu Zhang, Qiang Liu, and Shu Wu. 2022. A survey on deep graph generation: Methods and applications. In *Learning on Graphs Conference*, pages 47–1. PMLR.

A Appendix

A.1 Graph Structural Properties

- **# of Nodes and Edges**: the number of nodes and edges in the graph. It defines the basic size and density of the graphs.
- **Average Degree** (D^{Avg}): the average number of node degrees in the graph. Once the number of nodes and edges is determined, the average degree is also determined.
- **Max Degree** (D^{Max}): the maximum number of node degrees in the graph.
- **Clustering Coefficient** (CC): the measure of the degree to which nodes in a graph tend to cluster together. We first calculate the local clustering coefficient of each node, which quantifies how close its neighbors are to being a clique. The calculation is as follows:

$$CC_i = \frac{2\triangle_i}{D_i(D_i - 1)}, \tag{1}$$

  followed by the average operation across all nodes to calculate the network-level clustering coefficient. Specifically, we use *nx.average_clustering* API from NetworkX.
- **Triangle** (Tri): the total number of triangles contained in a network. Specifically, we first calculate the number of triangles involving each node using *nx.triangles* API from NetworkX and divide the total number by three.
- **Square** (Squ): the total number of squares contained in a network. As there is no well-established API to calculate this metric, we take inspiration from (Vignac et al., 2022) and calculate it as:

$$\text{Squ} = \frac{1}{8}(\text{Tr}(\mathbf{A}^4) - 2\sum_{v_i \in \mathcal{V}} d_i^2 + \sum_{v_i \in \mathcal{V}} d_i)$$

A.2 Prompt Examples and Generated Codes

Here, we show two examples of prompts and their corresponding codes generated by LLMs, one for T2G and the other one for T2G$^+$. Comparing these two, the key difference is in addition to the specified properties, T2G$^+$ also specifies the concrete methods to realize those properties based on domain expertise from network science.

Example Prompt for T2G

System Message: You are a network generator who is using the Python package NetworkX to write Python code to generate a network with the user-specified property. Please write a code to generate an undirected network with the following properties.

Properties:
- [Average Degree]: 3.9
- [Clustering of Coefficient]: 0.24
- [Number of Nodes]: 2708
- [Number of Triangles]: 1630.0
- [Maximum of Degree]: 168
- [Number of Squares]: 4664.0

Instructions:
- do not use an Erdos Renyi or Gnm graph
- make sure to generate a graph that closely matches ALL above [Properties]
- if the target property is not specified, do not modify the graph based on that property
- do not initialize the graph with no edges
- select the initial graph generator based on the domain and values of the properties requested
- do not include any explanations in the outputs
- make sure code is executable
- setup the maximum iteration to 10000
- setup the threshold of relative difference between the ground-truth and generated graph to be 1%

```
import networkx as nx
import pickle as pkl
import random

def generate_graph():
    # Initialize with a Barabasi-Albert graph to get a good starting point
    G = nx.barabasi_albert_graph(2708, 2)

    # Function to calculate the average degree
    def average_degree(G):
        return sum(dict(G.degree()).values()) / float(len(G))

    # Function to calculate the clustering coefficient
    def clustering_coefficient(G):
        return nx.average_clustering(G)

    # Function to calculate the number of triangles
    def number_of_triangles(G):
        return sum(nx.triangles(G).values()) / 3

    # Function to calculate the number of squares
    def number_of_squares(G):
        squares = 0
        for node in G:
            neighbors = list(G.neighbors(node))
            for i in range(len(neighbors)):
                for j in range(i + 1, len(neighbors)):
                    common_neighbors = set(G.neighbors(neighbors[i])).intersection(
    set(G.neighbors(neighbors[j])))
                    squares += len(common_neighbors)
        return squares / 4

    # Function to calculate the maximum degree
    def maximum_degree(G):
        return max(dict(G.degree()).values())

    # Target properties
```

```
    target_avg_degree = 3.9
    target_clustering_coefficient = 0.24
    target_triangles = 1630.0
    target_max_degree = 168
    target_squares = 4664.0

    # Iterative improvement
    max_iterations = 10000
    threshold = 0.01

    for _ in range(max_iterations):
        avg_degree = average_degree(G)
        clustering_coeff = clustering_coefficient(G)
        triangles = number_of_triangles(G)
        max_degree = maximum_degree(G)
        squares = number_of_squares(G)

        if (abs(avg_degree - target_avg_degree) / target_avg_degree < threshold and
            abs(clustering_coeff - target_clustering_coefficient) /
    target_clustering_coefficient < threshold and
            abs(triangles - target_triangles) / target_triangles < threshold and
            abs(max_degree - target_max_degree) / target_max_degree < threshold and
            abs(squares - target_squares) / target_squares < threshold):
            break

        # Randomly add or remove edges to improve properties
        if avg_degree < target_avg_degree:
            u, v = random.sample(G.nodes(), 2)
            if not G.has_edge(u, v):
                G.add_edge(u, v)
        else:
            u, v = random.choice(list(G.edges()))
            G.remove_edge(u, v)

    return G

G = generate_graph()
pkl.dump(G, open(f'./Cora/avg_degree_cc_tri_d_max_squ/graph_0.pkl', 'wb'))
```

Listing 1: LLM-generated Code with Prompting Scheme T2G

Example Prompt for T2G$^+$

System Message: You are a network generator who is using the Python package NetworkX to write Python code to generate a network with the user-specified property. Please write a code to generate an undirected network with the following properties.

Properties:
- [Average Degree]: 3.9
- [Clustering of Coefficient]: 0.24
- [Number of Nodes]: 2708
- [Number of Triangles]: 1630.0
- [Maximum of Degree]: 168
- [Number of Squares]: 4664.0

Methods:
- If [Average Degree] is specified, if it is lower than the target, add edges; if higher than the target, remove edges
- If [Maximum Degree] is specified, if it is lower than the target, add edges to the current max_degree node; if higher than the target, remove edges to the current max_degree node
- If [Clustering Coefficient] is specified, if it is lower than the target, randomly select a node and add edges within that node's neighbors; if higher than the target, randomly select a node and remove edges within that node's neighbors
- if [Number of Triangles] is specified, if it is lower than the target, randomly select a node and two of that node neighbors, and add an edge in-between; if higher than the target, randomly select a node and remove edges within that node's neighbors
- If [Number of Squares] is specified, randomly select a node and two of that node's neighbors; if lower than the target, randomly select another node to connect with those two neighbors; if higher than the target, find the intersection between neighbors of those two neighbors and remove edges to break the square. Note that we further provide the concrete code for generating Square as we empirically found LLMs are pretty difficult to understand Square structure.

Instructions:
- do not use an Erdos Renyi or Gnm graph
- make sure to generate a graph that closely matches ALL above [Properties]
- if the target property is not specified, do not modify the graph based on that property
- do not initialize the graph with no edges
- select the initial graph generator based on the domain and values of the properties requested
- do not include any explanations in the outputs
- make sure code is executable
- setup the maximum iteration to 10000
- setup the threshold of relative difference between the ground-truth and generated graph to be 1%

```
import networkx as nx
import numpy as np
import pickle as pkl
import random

def count_squares(G):
    A = nx.to_numpy_array(G)
    A4 = np.linalg.matrix_power(A, 4)
    trace_A4 = np.trace(A4)
    degrees = np.sum(A, axis=1)
    sum_deg_squared = np.sum(degrees ** 2)
    sum_deg = np.sum(degrees)
    squares_count = (trace_A4 - 2 * sum_deg_squared + sum_deg) / 8
    return squares_count

def generate_graph():
    G = nx.barabasi_albert_graph(2708, 2)
    target_avg_degree = 3.9
    target_clustering_coefficient = 0.24
    target_num_triangles = 1630.0
    target_max_degree = 168
    target_num_squares = 4664.0

    max_iterations = 10000
    threshold = 0.01

    for _ in range(max_iterations):
        avg_degree = sum(dict(G.degree()).values()) / G.number_of_nodes()
        clustering_coefficient = nx.average_clustering(G)
        num_triangles = sum(nx.triangles(G).values()) / 3
        max_degree = max(dict(G.degree()).values())
        num_squares = count_squares(G)
```

```
    if abs(avg_degree - target_avg_degree) / target_avg_degree < threshold and \
       abs(clustering_coefficient - target_clustering_coefficient) /
target_clustering_coefficient < threshold and \
       abs(num_triangles - target_num_triangles) / target_num_triangles <
threshold and \
       abs(max_degree - target_max_degree) / target_max_degree < threshold and \
       abs(num_squares - target_num_squares) / target_num_squares < threshold:
        break

    if avg_degree < target_avg_degree:
        u, v = random.sample(G.nodes(), 2)
        G.add_edge(u, v)
    elif avg_degree > target_avg_degree:
        u, v = random.choice(list(G.edges()))
        G.remove_edge(u, v)

    if max_degree < target_max_degree:
        max_node = max(G.degree, key=lambda x: x[1])[0]
        u = max_node
        v = random.choice(list(set(G.nodes()) - set(G.neighbors(max_node)) - {
max_node}))
        G.add_edge(u, v)
    elif max_degree > target_max_degree:
        max_node = max(G.degree, key=lambda x: x[1])[0]
        u, v = random.choice(list(G.edges(max_node)))
        G.remove_edge(u, v)

    if clustering_coefficient < target_clustering_coefficient:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
            G.add_edge(u, v)
    elif clustering_coefficient > target_clustering_coefficient:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
            if G.has_edge(u, v):
                G.remove_edge(u, v)

    if num_triangles < target_num_triangles:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
            G.add_edge(u, v)
    elif num_triangles > target_num_triangles:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
            if G.has_edge(u, v):
                G.remove_edge(u, v)

    if num_squares < target_num_squares:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
            w = random.choice(list(set(G.nodes()) - set(G.neighbors(u)) - set(G.
neighbors(v)) - {u, v}))
            G.add_edge(u, w)
            G.add_edge(v, w)
    elif num_squares > target_num_squares:
        node = random.choice(list(G.nodes()))
        neighbors = list(G.neighbors(node))
        if len(neighbors) >= 2:
            u, v = random.sample(neighbors, 2)
```

```
                common_neighbors = set(G.neighbors(u)).intersection(set(G.neighbors(
    v)))
                if common_neighbors:
                    w = random.choice(list(common_neighbors))
                    G.remove_edge(u, w)
                    G.remove_edge(v, w)

    return G

G = generate_graph()
pkl.dump(G, open(f'./Cora/avg_degree_cc_tri_d_max_squ/graph_0.pkl', 'wb'))
```

Listing 2: LLM-generated Code with Prompting Scheme T2G$^+$

```
import networkx as nx
import random
import numpy as np
import pickle as pkl

def count_squares(G):
    A = nx.to_numpy_array(G)
    A4 = np.linalg.matrix_power(A, 4)
    trace_A4 = np.trace(A4)
    degrees = np.sum(A, axis=1)
    sum_deg_squared = np.sum(degrees ** 2)
    sum_deg = np.sum(degrees)
    squares_count = (trace_A4 - 2 * sum_deg_squared + sum_deg) / 8
    return squares_count

def generate_graph():
    target_avg_degree = 2.15
    num_nodes = 13
    max_iterations = 10000
    threshold = 0.01

    G = nx.cycle_graph(num_nodes)
    current_avg_degree = sum(dict(G.degree()).values()) / num_nodes

    for _ in range(max_iterations):
        if abs(current_avg_degree - target_avg_degree) / target_avg_degree <
    threshold:
            break

        if current_avg_degree < target_avg_degree:
            u, v = random.sample(range(num_nodes), 2)
            if not G.has_edge(u, v):
                G.add_edge(u, v)
        else:
            u, v = random.choice(list(G.edges()))
            G.remove_edge(u, v)

        current_avg_degree = sum(dict(G.degree()).values()) / num_nodes

    return G

G = generate_graph()
pkl.dump(G, open(f'./MUTAG/avg_degree/graph_1.pkl', 'wb'))
```

Listing 3: LLM-generated Code for Generating MUTAG Graph by Initializing Cycles

A.3 Experimental Settings

Dataset: Here, we specify our experimental setting. For the dataset, we take graphs from these three domains: citation, social, and molecule datasets. For citation networks, we take Cora, Citeseer, and Pubmed from the well-established Planetoid. For social and molecule networks, we take IMDB-Binary, REDDIT-Binary, MUTAG, PROTEINS, NCI1, DD, ENZYME from the TUdataset. The statistics of each dataset are shown in Table 2. For each dataset, we select 50 graphs for evaluation to save the API-call budget.

Evaluation: For each dataset containing N graphs, we generate a corresponding set of N graphs by prompting LLMs to generate code based on the properties extracted from the original ground-truth graphs. We then compute the percentage difference for each metric as follows:

$$\alpha_{ij} = \frac{|p_{ij} - p'_{ij}|}{p_{ij}}, \tag{2}$$

where p_{ij}, p'_{ij} respectively represents the jth-property of the ith graph for the original ground-truth one and the generated one. α_{ij} represents the performance of the generated i^{th} graph for j^{th} metric. Furthermore, we average this performance across all graph instances to obtain the final performance ϕ_j for j^{th} metric.

$$\phi_j = 1 - \frac{1}{N}\sum_{i=1}^{N} \alpha_{ij}, \quad \forall j \in \Phi \tag{3}$$

where Φ denotes the set of metrics, including clustering coefficient (CC), number of nodes/edges, average degree (D^{Avg}), triangle (Tri), square (Squ), and a maximum of degree (D^{Max}).

LLMs: We select GPT4o (2024-05-13b) as our default code generator.

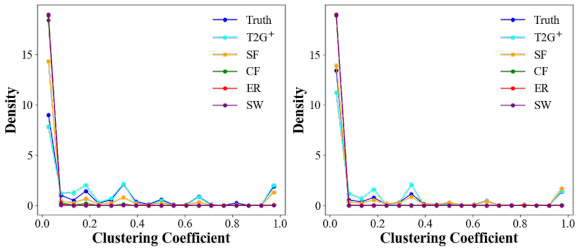
Baselines: For Table 1, we benchmark our proposed T2G/T2G$^+$ methods with conventional heuristic-based methods, which are:

- **Barabasi-Albert (BA)** (Barabási and Albert, 1999): A preferential attachment model that generates scale-free networks, where new nodes connect to existing high-degree nodes, resulting in a power-law degree distribution. This model captures the preferential attachment process observed in many real-world networks. We vary the number of edges each newly added node forms, ranging from 1 to twice the graph's average degree.
- **Small World (SW)** (Watts and Strogatz, 1998): A graph generation model that creates networks with high clustering and short average path lengths resembling real-world small-world properties. We employ the Watts-Strogatz algorithm, where each node is initially connected to its nearest neighbors based on the average degree of the ground-truth graph. The probability of rewiring edges is tuned from 0 to 1 in increments of 0.02 to introduce varying levels of randomness.
- **Erdős-Rényi (ER)** (Erdos et al., 1960): A random graph model where edges are formed between node pairs with a constant probability, resulting in a Poisson degree distribution. This model is commonly used to study random network structures. The edge formation probability is tuned from 0 to 1 with a 0.02 interval to generate a range of random graph structures.
- **Scale-Free (SF)** (Bollobás et al., 2003): A model that generates networks following a power-law degree distribution, similar to the Barabasi-Albert model but with more tunable parameters. These networks exhibit a few highly connected nodes or hubs. We tune the hyperparameters α, β, Δ_1, and Δ_2 from 0.1 to 1 in 0.2 intervals to capture different structural variations of scale-free networks.
- **Configuration (CF)** (Newman, 2003): A model that generates graphs based on a given degree sequence, ensuring that the resulting graph retains the same degree distribution as the input. This method preserves degree-related properties and is ideal for cases where maintaining the original degree distribution is important.

For each of the above baselines, we tune their corresponding hyperparameters to select the optimal ones by the performance of the generated graph (the average difference of the property between the generated graphs and ground-truth graphs).

Table 2: Dataset Statistics. CC - Clustering Coefficient, D^{Avg} - Average Degree, Tri - Triangle, Squ - Square, D^{Max} - Maximum Degree.

Dataset	CC	# Nodes	# Edges	D^{Avg}	Tri	Squ	D^{Max}
MUTAG (MUT)	0.0 ± 0.0	17.76 ± 4.16	19.7 ± 5.2	2.2 ± 0.1	0.0 ± 0.0	0.0 ± 0.0	3.02 ± 0.14
PROTEINS (PROT)	0.45 ± 0.2	62.78 ± 85.22	120.84 ± 180.35	3.69 ± 0.42	49.9 ± 97.96	78.02 ± 165.91	5.96 ± 1.02
DD	0.48 ± 0.04	520.2 ± 939.98	1305.82 ± 2333.0	5.04 ± 0.48	899.1 ± 1524.75	1774.14 ± 2897.8	9.68 ± 1.19
ENZYMES (ENZY)	0.53 ± 0.19	34.94 ± 16.76	66.06 ± 27.96	3.84 ± 0.77	32.46 ± 14.05	47.46 ± 25.19	5.88 ± 1.32
NCI1	0.0 ± 0.0	23.72 ± 6.77	25.14 ± 7.55	2.11 ± 0.09	0.0 ± 0.0	0.0 ± 0.0	3.16 ± 0.37
IMDB-B (IMDB)	0.95 ± 0.03	21.66 ± 12.04	100.72 ± 64.06	9.32 ± 3.7	307.36 ± 334.89	2585.98 ± 4203.62	20.66 ± 12.04
REDDIT-B (REDDIT)	0.04 ± 0.02	550.42 ± 416.89	642.24 ± 488.03	2.31 ± 0.13	27.82 ± 23.97	115.06 ± 154.26	279.78 ± 242.51
Cora	0.24 ± 0.0	2708.0 ± 0.0	5278.0 ± 0.0	3.9 ± 0.0	1630.0 ± 0.0	4664.0 ± 0.0	168.0 ± 0.0
Citeseer (Cite)	0.14 ± 0.0	3327.0 ± 0.0	4552.0 ± 0.0	2.74 ± 0.0	1167.0 ± 0.0	6060.0 ± 0.0	99.0 ± 0.0
Pubmed (Pub)	0.06 ± 0.0	19717.0 ± 0.0	44324.0 ± 0.0	4.5 ± 0.0	12520.0 ± 0.0	163505.0 ± 0.0	171.0 ± 0.0



(a) Cora **(b) Citeseer**

Figure 5: Visualization of the generated and ground-truth degree distribution for Cora (a) and Citeseer (b).

A.4 Comprehensive Results

Here we present comprehensive results to complement the result analysis in Section 4.

Clustering Coefficient Distribution Analysis: Instead of comparing the global differences in clustering coefficients between the generated graph and the ground-truth graph, we perform a local comparison by visualizing their clustering coefficient distributions in Figure 5(a) and (b) for the Cora and Citeseer datasets. We find that only SF and T2G$^+$ can replicate the distribution of the clustering coefficient of the original network.

Generation Performance on Other Datasets: In Figure 6-13, we visualize the graph generation performance on additional datasets. Here, we observe that T2G$^+$ performs significantly better than T2G due to the incorporation of additional domain expertise. Furthermore, incorporating high-order graph structural properties leads to lower performance, which aligns with our observations in the main text.

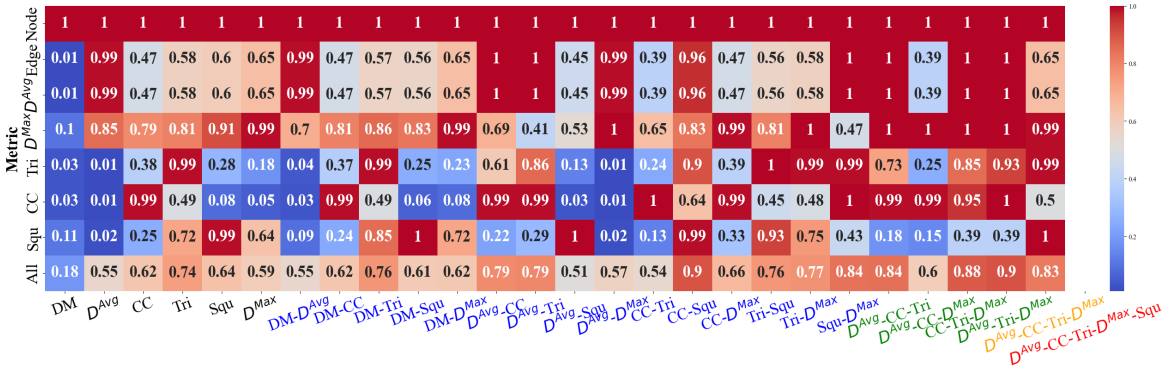


Figure 6: T2G$^+$ Performance of generating citation network Cora with prescribed structural properties.

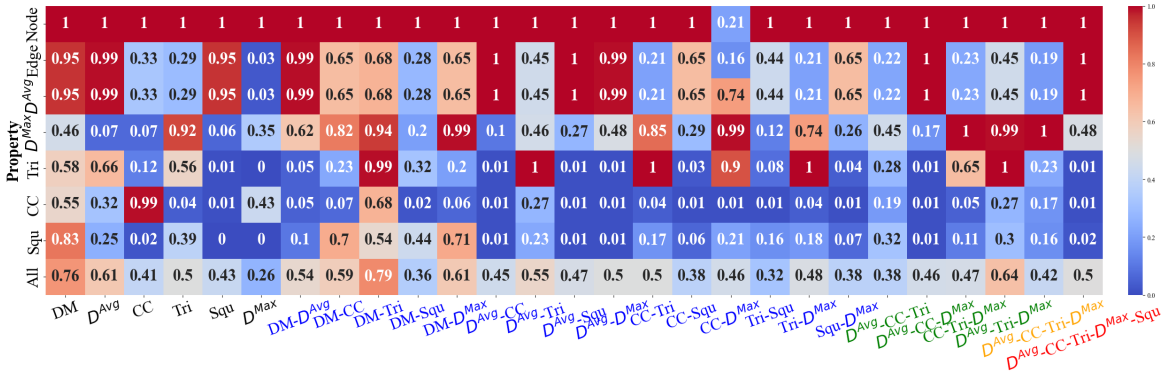


Figure 7: T2G Performance of generating citation network Cora with prescribed structural properties.

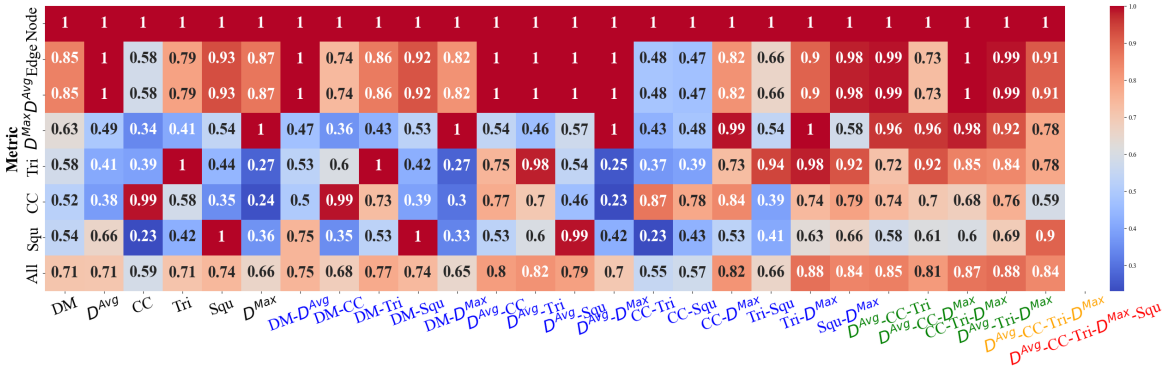


Figure 8: T2G$^+$ Performance of generating molecular network Protein with prescribed structural properties.

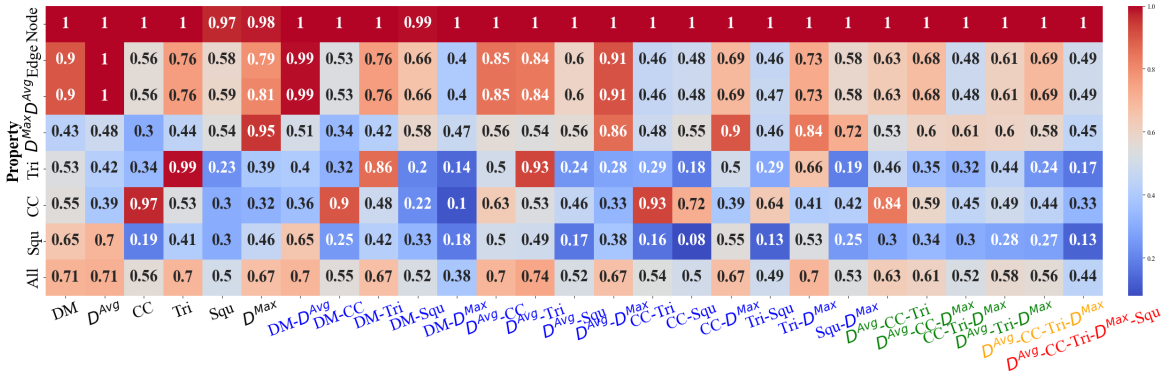


Figure 9: T2G Performance of generating molecular network Protein with prescribed structural properties.

Figure 10: T2G$^+$ Performance of generating social network IMDB with prescribed structural properties.

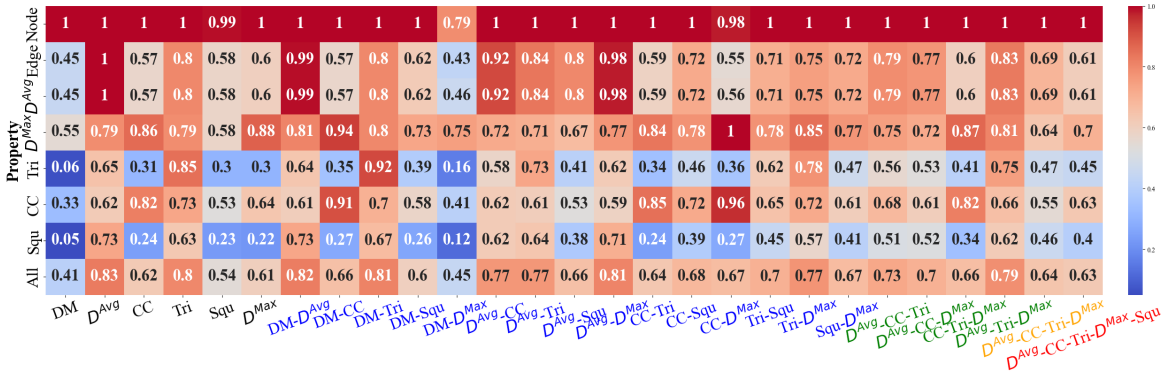


Figure 11: T2G Performance of generating social network IMDB with prescribed structural properties.

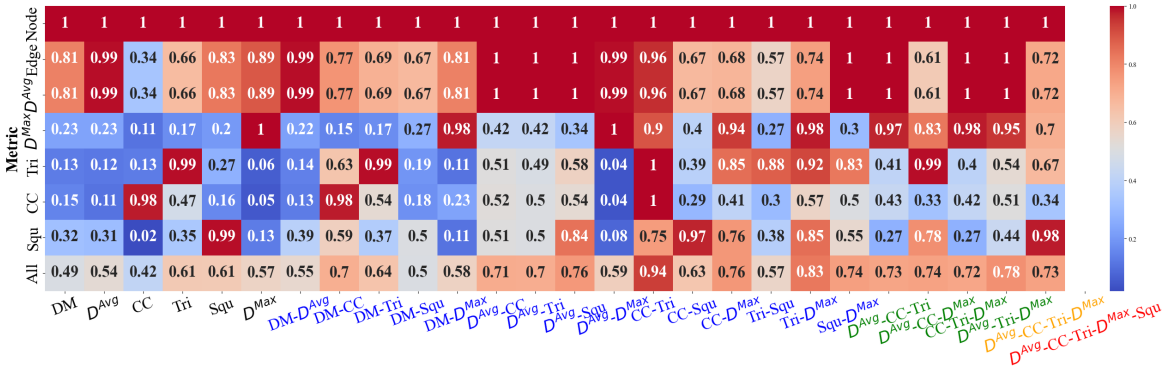


Figure 12: T2G$^+$ Performance of generating molecular network DD with prescribed structural properties.

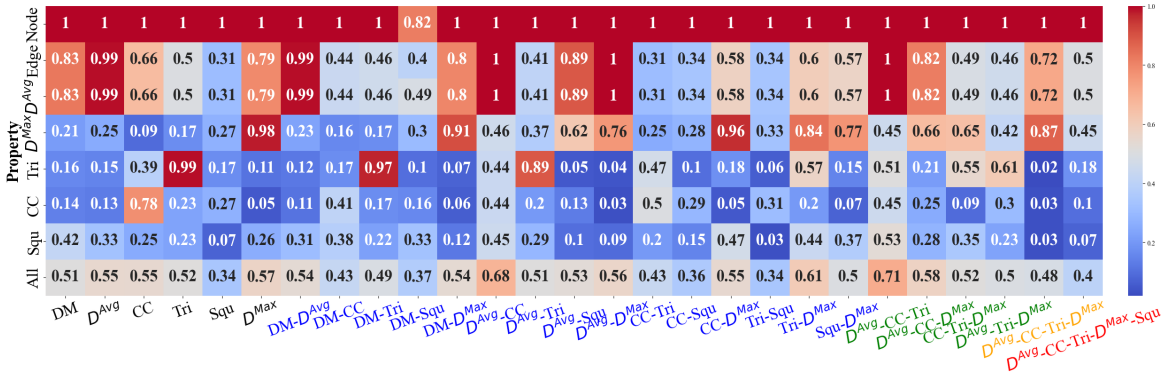


Figure 13: T2G Performance of generating molecular network DD with prescribed structural properties.